Solfec Performance Assessment Report

Document Information

Reference Number	POP_AR_83
Author	Nick Dingle (NAG)
Contributor(s)	Wadud Miah (NAG), Jon Gibson (NAG)
Date	11/12/17

Notices: *The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 676553.*

Contents

1 Background 3

2 Application Structure 3

3 Region of Interest (RoI) 3

4 Scalability 5

5 Efficiency 5

6 Load Balance 6

7 Computational Performance 7

8 Communications 8

9 Summary of Observations 9

References 11

1 Background

Applicants Name: Tomasz Koziara
Application Name: Solfec
Programming Languages: C and Python
Programming Model: MPI
Source Code Available: Yes
Input data: `array-of-cubes.py`
Performance study: General code audit

The execution traces used in this audit were gathered on CINECA's *Marconi* machine, which is a Lenovo NeXtScale cluster containing 1 512 nodes in 21 racks. Each node has 2x 18-core Intel Xeon E5-2697 v4 (Broadwell) 2.30GHz CPUs with 128GB of RAM and is connected by a 100Gb/s Intel Omnipath network. The 54 432 cores have a total peak performance of 2 PFlop/s.

The `array-of-cubes.py` example is a finite element multibody structural problem in which a stack of cubes is subject to a sine sweep acceleration signal. We set the array edge size parameter, `M`, to 32 and the cube mesh edge size parameter, `N`, to 4. This generates 32 774 bodies, 12 288 036 degrees of freedom and approximately 585 000 constraints. These values are representative of the size of problems currently analysed by Solfec users.

Traces were recorded using Extrae 3.4.3 and analysed using Paraver 4.6.3. Due to the amount of trace data generated, especially at large core counts, we used Extrae's API to limit collection to only the Region of Interest defined in Section 3.

2 Application Structure

Solfec is a computational contact dynamics code written in C and Python and parallelised with MPI. It also incorporates several third-party codes written in C, C++ and Fortran. Solfec includes mesh, convex polyhedra, sphere and ellipsoid based shapes, linear elastic first order finite elements, pseudo-rigid and rigid kinematics, velocity-based Signorini-Coulomb contact/impact law, and parallel time stepping combined with a simple dynamic load balancing. For this audit we compiled Solfec with Intel MKL for BLAS and LAPACK routines [1], Zoltan for load balancing [2], and HDF5 for IO [3].

3 Region of Interest (RoI)

The Region of Interest (RoI) is 2 timesteps from the middle of the simulation, and Figure 1 shows a timeline of its execution on 1 node (36 cores). There are two phases in each timestep:

1. A long period of computation to update the domain and local dynamics (coloured blue).
2. The evaluation of the constraints using an iterative Newton solver. This is visible as more frequent communications (coloured orange) interspersed with short periods of computation (coloured blue).

The first timestep contains 7 iterations of the Newton solver; the second contains 8.

Figure 2 shows the timeline of the useful duration (actual computation) in the RoI on 1 node (36 cores) and Figure 3 shows the corresponding MPI call timeline. Figure 2 is coloured depending on the length of time spent in computation, from short (green) to longer times (blue).

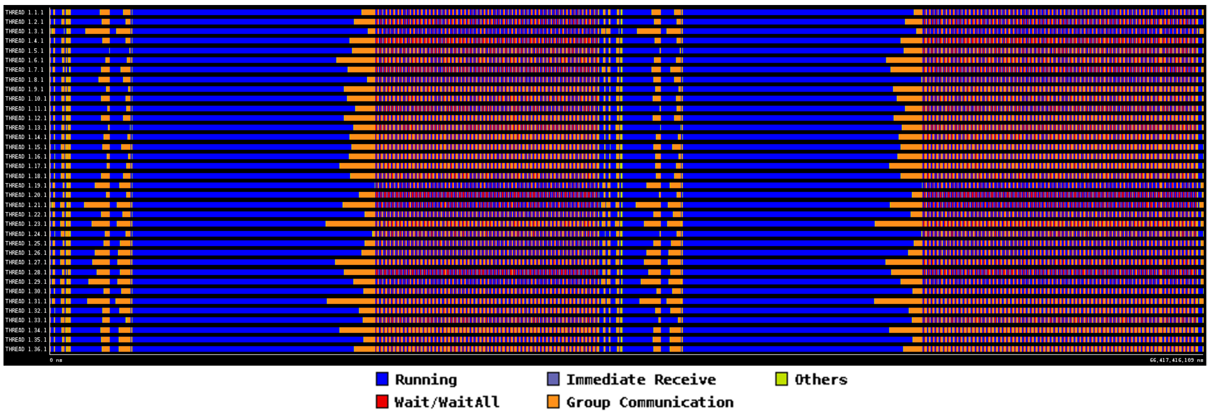


Figure 1: Timeline of the Region of Interest on 1 node (36 cores).

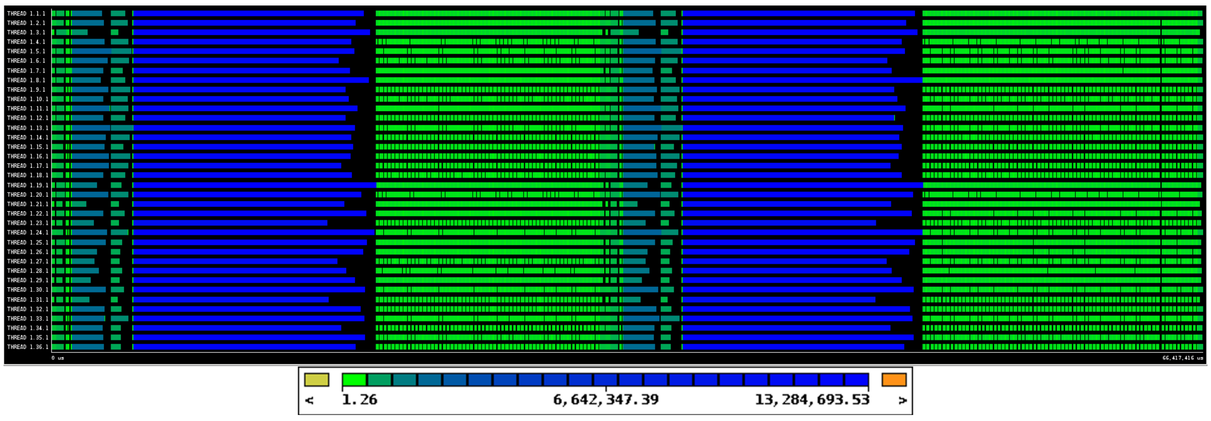


Figure 2: Useful duration timeline of the Region of Interest on 1 node (36 cores).

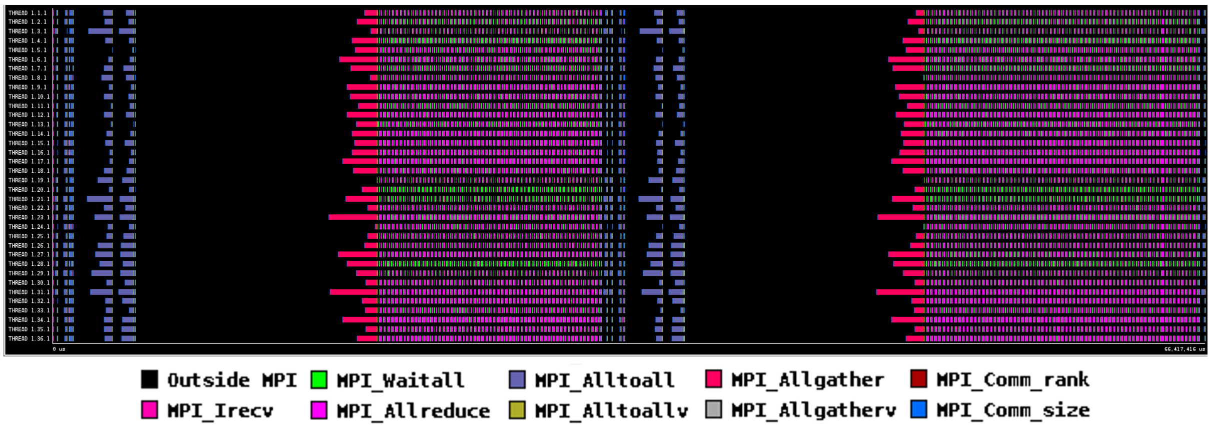


Figure 3: MPI call timeline of the Region of Interest on 1 node (36 cores).

The structure of the timesteps and Newton iterations is clearly visible, with the two phases within a timestep delineated by an `MPI_Allgather`. Communication in the first phase is largely confined to `MPI_Alltoall`, while the Newton solver uses `MPI_Irecv`, `MPI_Isend`, `MPI_Waitall` and `MPI_Allreduce`.

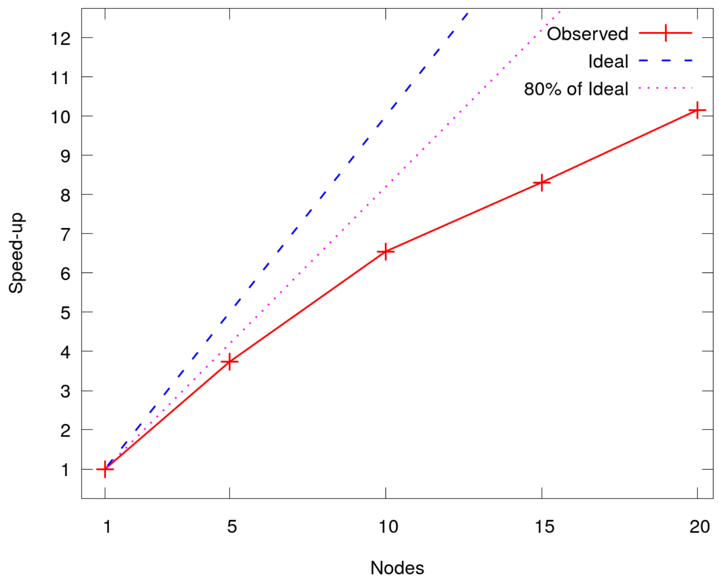


Figure 4: Speed-up of the Region of Interest.

	# Nodes				
	1	5	10	15	20
Global Efficiency	0.81	0.60	0.53	0.45	0.41
Computational Efficiency	1.00	0.98	0.96	0.88	0.87
Parallel Efficiency	0.81	0.61	0.55	0.51	0.47
Load Balance	0.89	0.79	0.74	0.73	0.65
Communication Efficiency	0.91	0.77	0.74	0.70	0.71
Serialisation Efficiency	0.93	0.94	0.87	0.81	0.81
Transfer Efficiency	0.98	0.82	0.85	0.87	0.88
IPC Efficiency	1.00	1.06	1.10	1.06	1.10
Instructions Efficiency	1.00	0.93	0.87	0.83	0.80

Table 1: Efficiency metrics for the Region of Interest.

4 Scalability

Figure 4 shows the speed-up of the RoI on up to 20 nodes (720 cores). We observe that speed-up increases with increasing node count, but that it remains below 80% of the ideal.

5 Efficiency

We quantify the performance of the RoI using the metrics presented in Table 1. The values are efficiencies that generally range from 0 to 1, with 1 being the ideal and 0.8 considered the cut-off for 'good' performance. We define *useful time* to be time within computation only, e.g. excluding MPI and IO, and *useful instructions* therefore refers to instructions executed within computational regions only. The efficiencies are defined as follows:

- Global Efficiency is the product of Computational Efficiency and Parallel Efficiency.

- Computational Efficiency shows how the total time spent in computation varies with the number of processes. It is a relative scaling of total time in computation compared to the value on the smallest number of MPI processes.
- Parallel Efficiency is the product of Load Balance, Serialisation and Transfer Efficiencies.
- Load Balance is the ratio of the average time processes spend in computation to the maximum time.
- Communication Efficiency is the product of Transfer and Serialisation Efficiencies.
- Serialisation Efficiency measures the overhead incurred from processes waiting for other communication partners to arrive.
- Transfer Efficiency gives the amount of time lost to transferring data.
- Instruction Efficiency compares the total number of useful instructions executed for different numbers of processes relative to the value on the smallest number of MPI processes.
- IPC is the number of useful instructions executed per cycle and the IPC Efficiency shows how its value varies with process count. In Table 1, the values of computational, instructions and IPC efficiencies are all calculated relative to the single node performance.

A more detailed description of these metrics can be found on the POP website [4].

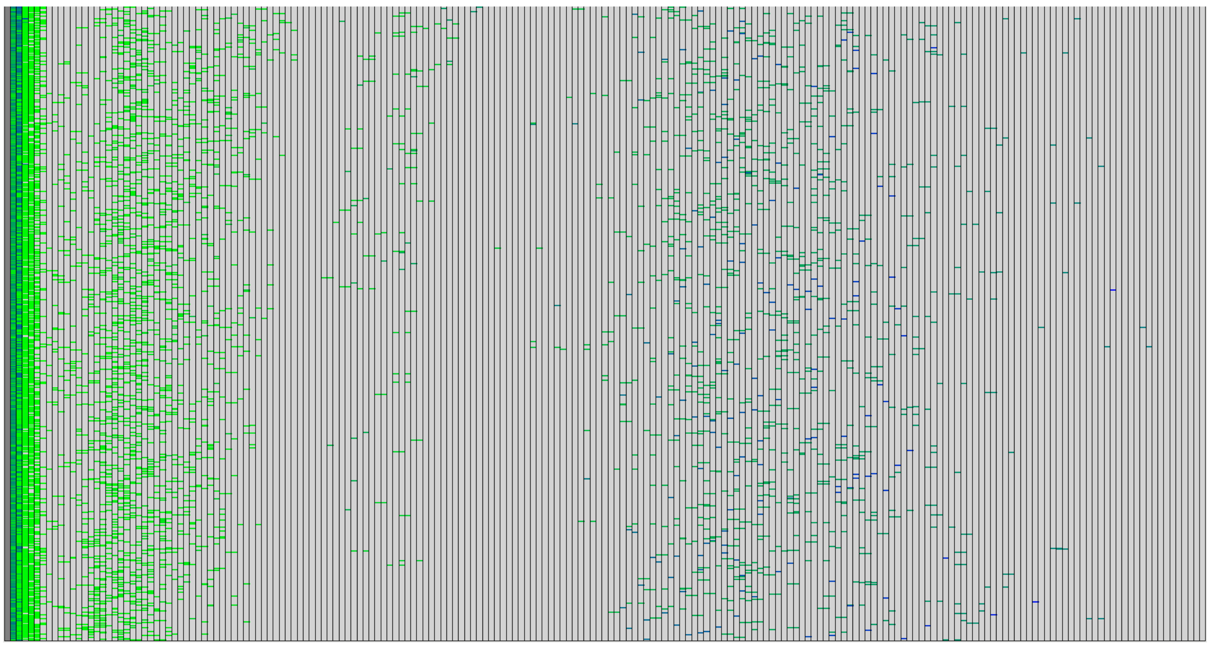
We observe that the main source of inefficiency is Load Balance, which has dropped to 0.65 on 20 nodes (740 cores). Communication Efficiency is also low, which suggests that the code is spending a large amount of time in MPI calls; we will investigate this in more detail Section 8. Serialisation Efficiency and Transfer Efficiency both remain above 0.8 in all cases, although Serialisation Efficiency has declined to 0.81 on 20 nodes. Interestingly Transfer Efficiency drops from 0.98 to 0.82 when going from 1 node to 5 nodes, but then increases again with increasing core count.

The decline in Computational Efficiency is due to the falling Instruction Efficiency, which reflects the fact that the total number of instructions executed grows as the number of nodes is increased. This suggests that there may be some replication of calculations across processes. IPC is actually higher on multiple node runs than on 1, and this somewhat offsets the rise in the number of instructions. This rise in IPC may be because the smaller problem sizes on each process at higher process counts are more cache efficient.

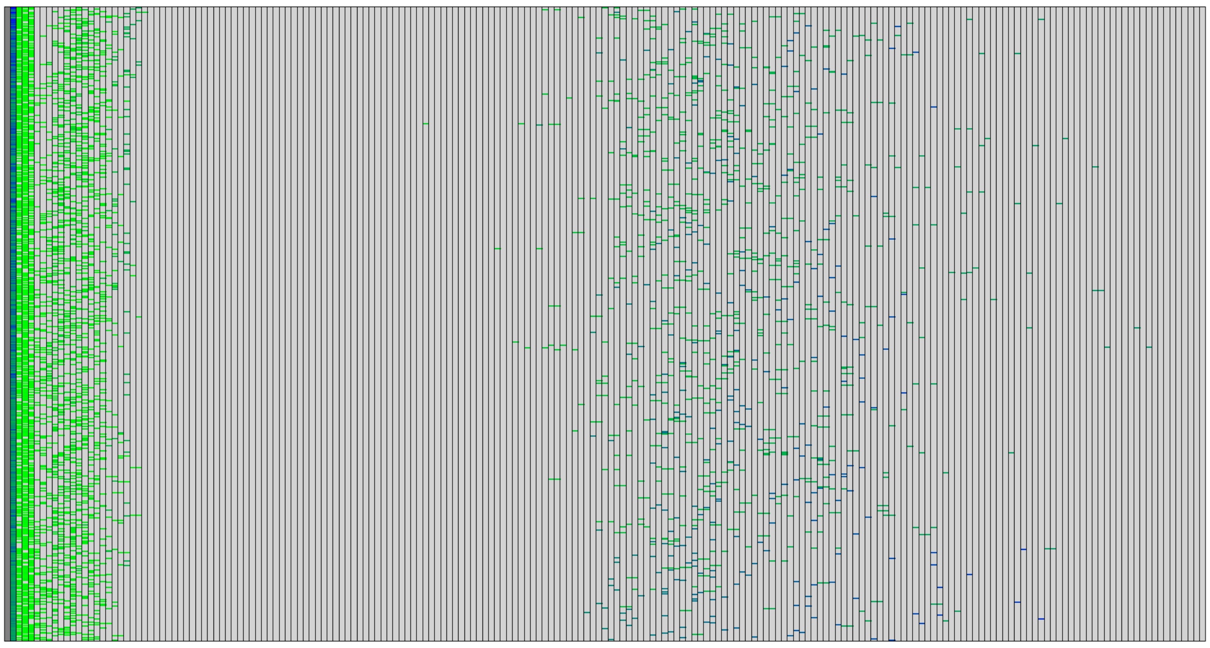
6 Load Balance

Section 5 identified that Load Balance was a major source of inefficiency. To investigate to what extent this can be attributed to the way in which work is divided across processes, we plot histograms of instructions and useful duration in Figure 5. These display binned values for the two metrics, with darker colours signifying that the code had more of those periods of computation or more often executed that number of instructions. The y-axis is processes and the x-axis is bin values.

We observe that there is considerable variation in the number of instructions executed and in the corresponding durations of periods of computation. This suggests that the decline in Load Balance Efficiency can be attributed to increasingly unbalanced computational load. Further work could be conducted in a POP Performance Plan or Proof-of-Concept to identify the routines that suffer most from load imbalance and to try different approaches (e.g. different Zoltan parameters or other load balancing tools) to see if they lead to any improvement.



(a) Useful duration



(b) Instructions

Figure 5: Useful duration and instructions histograms on 20 nodes (720 cores).

7 Computational Performance

Table 1 contains two metrics investigating how well the computational aspects of the code scale: IPC Efficiency and Instructions Efficiency. Table 2 presents the corresponding absolute values of IPC and number of instructions. Ideally the number of instructions should remain constant when the number of processes increases because we are strong-scaling the problem under analysis, but here we see that the number of instructions actually increases with the

	# Nodes				
	1	5	10	15	20
IPC	1.15	1.22	1.26	1.22	1.27
Instructions ($\times 10^6$)	4.79	5.15	5.48	5.79	6.00

Table 2: Absolute values of IPC and number of instructions.

	# Nodes				
	1	5	10	15	20
`MPI_Allreduce`	9.4%	18.9%	18.1%	22.4%	20.8%
`MPI_Alltoall`	4.0%	5.6%	7.0%	9.6%	12.7%
`MPI_Allgather`	3.8%	6.6%	9.6%	9.0%	9.8%
`MPI_Waitall`	2.0%	6.5%	7.5%	5.6%	7.1%

Table 3: Average percentage of runtime of MPI calls in the RoI. We only list routines that consume >1% runtime.

	# Nodes				
	1	5	10	15	20
`MPI_Allreduce`	8.2	8.2	8.2	8.2	8.2
`MPI_Alltoall`	12	12	12	12	12
`MPI_Allgather`	4	4	4	4	4
`MPI_Waitall`	188 042.5	106 568.5	82 522.7	71 301.3	62 056.1

Table 4: Average buffer size in bytes per MPI call on each process for the most time-consuming routines shown in Table 3.

number of processes. A common cause of growth in the number of instructions is replication of computation across processes, and in this audit there may also be overheads from using Zoltan. We also observe that IPC improves as the number of processes increases. As hypothesised in Section 5, this may be because the cache efficiency improves at higher process counts.

The reasons for the growth in the number of instructions could be investigated in a POP Performance Plan. We could use approaches such as clustering or timeline analysis with Extrae and Paraver to identify areas of the code which experience the largest rises in instructions, and use this information to target further optimisations.

8 Communications

Table 3 shows the percentages of runtime spent in MPI calls in the RoI. `MPI_Allreduce` is the most time-consuming MPI routine, taking up approximately 20% of the RoI's runtime from the 5 node execution onward. This routine is used in the iterative Newton solver.

Table 4 shows that the average buffer sizes in bytes per `MPI_Allreduce` call on each process is constant with increasing node counts. As each process is contributing only a single double-precision floating point number (8 bytes) to the reduction, we suspect that the majority of time in the routine is spent waiting for late arrivals caused by local computational load imbalance rather than actually transferring the data.

We investigate this supposition by examining the performance of the code on a simulated ideal network with zero latency and infinite bandwidth. Figure 6 compares the timelines of MPI calls in the RoI on 20 nodes (720 cores) on the real and ideal networks. We see that the

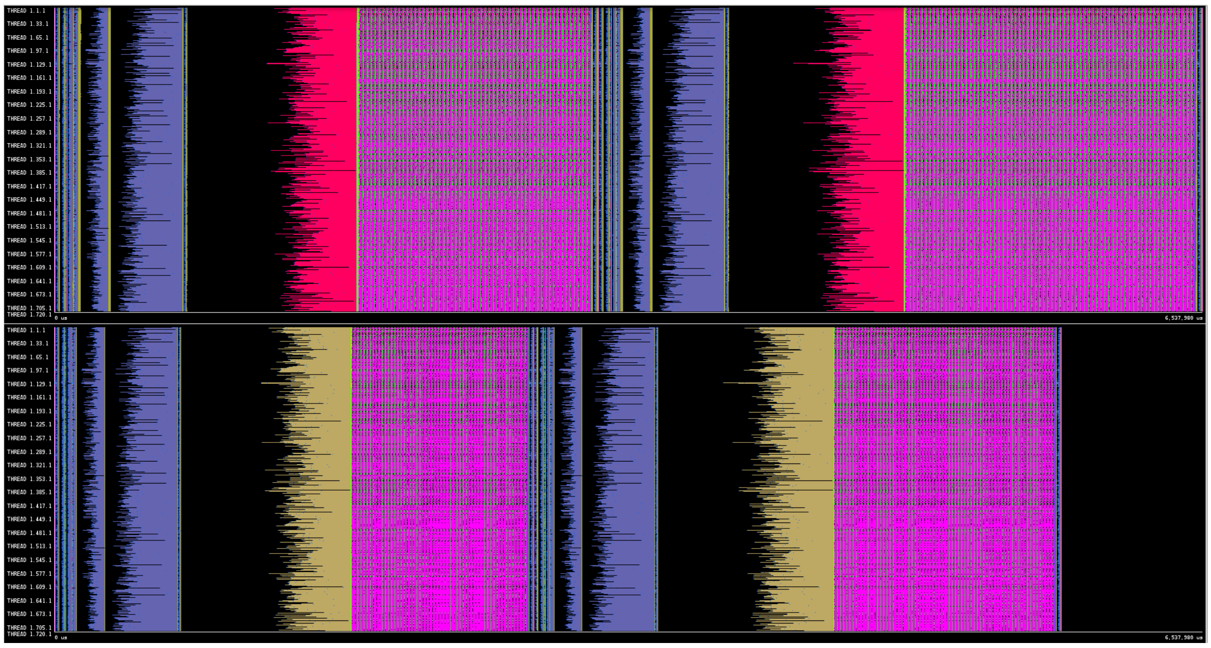


Figure 6: Original (top) and simulated ideal (bottom) MPI call timeline of the Region of Interest on 20 nodes (720 cores).

amount of time in MPI is not greatly reduced by the ideal network, which suggests that the time required to transfer the data is not the dominant factor. This conclusion is supported by the relatively good values for Transfer Efficiency in Table 1.

It would be interesting to see whether Solfec's performance could be improved by replacing `MPI_Allreduce` with `MPI_Iallreduce`/`MPI_Wait`. This work could be accomplished in a POP Proof-of-Concept study. If the source of the `MPI_Allreduce` calls is the HYPRE library [5] (which Solfec may use within its Newton solver) then it could be worth investigating the scalability of other solvers. For example, the PETSc library [6] includes an implementation of pipelined GMRES that replaces `MPI_Allreduce` with `MPI_Iallreduce`/`MPI_Wait` [7].

9 Summary of Observations

We observed that the main source of inefficiency in Solfec was Load Balance. The histograms in Section 6 show that there was considerable variation between processes in the numbers of instructions executed during computational portions of the code. We also saw that the total number of instructions executed grew as the number of MPI processes increased. Additionally, Solfec spent about 20% of its time in `MPI_Allreduce` when running on more than 1 node.

We recommend that further studies should be conducted to:

- Ascertain which routines suffer most from load imbalance and investigate whether changing the partitioning parameters or load balancing tool leads to any improvement. The version of Solfec used in this audit uses Zoltan for load balancing, and various control parameters are passed through multiple calls to `Zoltan_Set_Param()`. Changing the values of these parameters may improve the load balance. Solfec can be built to use Dynlb [8] instead of Zoltan, and so the potential improvement that this offers could also be assessed.

- Identify the root cause of the growth in number of instructions executed. This will require identifying the code regions that are most responsible for the increase in instructions (using further tracing and analysis) and using this information to guide future optimisations.

- Investigate whether the Solfec's performance could be improved by replacing `MPI_Allreduce` with `MPI_Iallreduce` and `MPI_Wait`. This might be achieved by using a different iterative solver such as pipelined GMRES.

References

[1] Intel Math Kernel Library. https://software.intel.com/en-us/mkl.

[2] Erik Boman, Karen Devine, Lee Ann Fisk, Robert Heaphy, Bruce Hendrickson, Vitus Leung, Courtenay Vaughan, Ümit Çatalyürek, Doruk Bozdag, and William Mitchell. Zoltan home page. http://www.cs.sandia.gov/Zoltan.

[3] The HDF Group. Hierarchical Data Format, version 5, 1997-2017. http://www.hdfgroup.org/HDF5/.

[4] Efficiency metrics in a POP performance audit. https://sharepoint.ecampus.rwth-aachen.de/units/rz/HPC/public/Shared%20Documents/Metrics.pdf.

[5] HYPRE: Scalable linear solvers and multigrid methods. https://computation.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods/software.

[6] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc web page. http://www.mcs.anl.gov/petsc.

[7] Ichitaro Yamazaki, Mark Hoemmen, Piotr Luszczek, and Jack Dongarra. Improving performance of GMRES by reducing communication and pipelining global collectives. In *Proceedings of the 18th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2017)*, Orlando FL, June 2017.

[8] Dynlb: a minimalist dynamic load balancer for points in 3D. https://github.com/tkoziara/dynlb.